实验四 电影评论情感分类

学号：202228019427035　　姓名：赵巍山　　培养单位：空天信息创新研究院

一、 实验目的

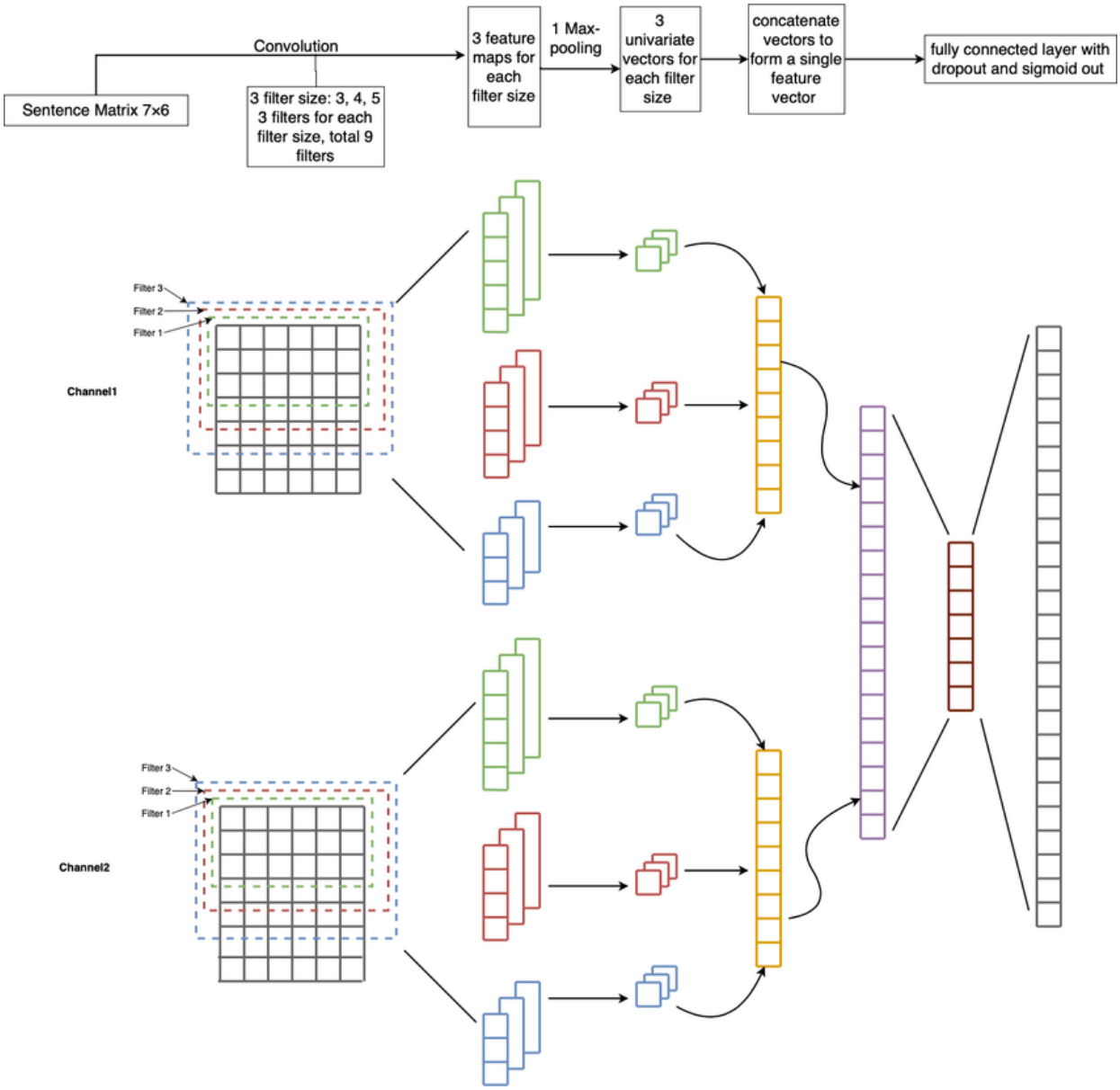
1. 进一步加深对卷积神经网络基本原理的理解。
2. 掌握卷积神经网络处理文本的各项技术。
3. 掌握文本分类模型 Text-CNN 的架构和原理。

二、 实验要求

1. 任选一个深度学习框架建立Text-CNN模型（本实验指导书以TensorFlow为例）。
2. 实现对中文电影评论的情感分类，实现测试准确率在83%以上。
3. 也可采用LSTM实现，实现测试准确率高于卷积神经网络。
4. 按规定时间在课程网站提交实验报告、代码以及PPT。

三、 实验原理

Text-CNN和传统的CNN结构类似，具有词嵌入层、卷积层、池化层和全连接层的四层结构，如图1所示。



如图1中所示，Text-CNN的词嵌入层（Word embedding）使用二维矩阵来表示长文本。词嵌入将输入文本的每个词语通过空间映射，将独热表示（One-Hot Representation）转换成分布式表示（Distributed Representation），进而可以使用低维的词向量来表示每一个词语。经过词嵌入，每个单词具有相同长度的词向量表示。将各个词语的向量表示连起来便可以得到二维矩阵。得到词向量的方式有多种，常用的是Word2vec方法。若使用预训练好的词向量，在训练模型的时候可以选择更新或不更新词向量，分别对应嵌入层状态为Non-static和Static。

Text-CNN的卷积层是主要部分，卷积核的宽度等于词向量的维度，经卷积后可以提取文本的特征向量。与在图像领域应用类似，Text-CNN可以设置多个卷积核以提取文本的多层特征，长度为N的卷积核可以提取文本中的N-gram特征。

Text-CNN的池化层一般采取Max-over-time pooling，输出最大值，从而判断词嵌入中是否含N-gram。

Text-CNN的全连接层采用了Dropout算法防止过拟合，并使用Softmax函数输出各个类别的概率。

四、实验所用工具及数据集

1. **主要工具**

   Python 3.6.9、PyTorch 1.4.0、numpy-1.16.6、gensim-4.0.1

2.数据集：

1. 训练集。包含19998条中文电影评论，其中正负向评论各9999条。
2. 验证集。包含5629条中文电影评论，其中正负向评论各2812,2817条。
3. 测试集。包含369条中文电影评论，其中正负向评论各187,182条。
4. 预训练词向量。中文维基百科词向量word2vec：wiki_word2vec_50.bin。

五、实验步骤：

加载实验所需的函数库：

```
import torch
import torch.nn as nn
from torchvision import transforms, datasets, utils
import matplotlib.pyplot as plt
import numpy as np
import torch.optim as optim
from tqdm import tqdm
import torch.optim as optim
from mymodel import TextCNN
import gensim
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from collections import Counter
from torch.utils.data import TensorDataset, DataLoader
```

数据预处理:

从train.txt, validation.txt构建词汇表并存储，形如{word: id}

```
def build_word2id(save_to_path=None):

    word2id = {'_PAD_': 0}
    path = ['./Dataset/train.txt', './Dataset/validation.txt']

    # write the index to word2id[word]
    for _path in path:
        with open(_path, encoding='utf-8') as f:
            for line in f.readlines():
                sp = line.strip().split()
                for word in sp[1:]:
                    if word not in word2id.keys():
```

```
                        word2id[word] = len(word2id)
    if save_to_path:
        with open(save_to_path, 'w', encoding='utf-8') as f:
            for w in word2id:
                f.write(w+'\t')
                f.write(str(word2id[w]))
                f.write('\n')

    return word2id
########基于预训练好的 word2vec 构建训练语料中所含词语的
word2vec###############################################3
def build_word2vec(fname, word2id, save_to_path=None):
    """
    :param fname:预训练的 word2vec
    :param word2id: 语料文本中包含的词汇集
    :param save_to_path:# 保存训练语料库中的词组对应的 word2vec 到本地
    :return:语料文本中词汇集对应的 word2vec 向量{id: word2vec}
    """
    n_words = max(word2id.values()) + 1
    model = gensim.models.KeyedVectors.load_word2vec_format(fname, binary=True)
    wordid_vecs = np.array(np.random.uniform(-1., 1., [n_words, model.vector_size]))
    for word in word2id.keys():
        try:
            wordid_vecs[word2id[word]] = model[word]
        except KeyError:
            pass
    if save_to_path:
        with open(save_to_path, 'w', encoding='utf-8') as f:
            for vec in wordid_vecs:
                vec = [str(w) for w in vec]
                f.write(' '.join(vec))
                f.write('\n')
    return wordid_vecs
```

定义参数的初始化：

```
#################定义初始化的参数
###########################################################
class CONFIG():
    update_w2v = True           # # 是否在训练中更新 w2v
    vocab_size = 58954          #词汇量，与 word2id 中的词汇量一致
    n_class = 2                 # 分类数：分别为 pos 和 neg
    embedding_dim = 50          # 词向量维度
    drop_keep_prob = 0.5        # dropout 层，参数 keep 的比例
    num_filters = 256           #  卷积层 filter 的数量
    kernel_size = 3             # 卷积核的尺寸
    pretrained_embed =
build_word2vec('./Dataset/wiki_word2vec_50.bin',build_word2id('./Dataset/word2id.txt
'))
```

定义模型：

```
import torch.nn as nn
import torch.nn.functional as F
import torch

#定义我们的模型
class TextCNN(nn.Module):
    def __init__(self, config):
        super(TextCNN, self).__init__()
        update_w2v = config.update_w2v
        vocab_size = config.vocab_size
        n_class = config.n_class
        embedding_dim = config.embedding_dim
        num_filters = config.num_filters
        kernel_size = config.kernel_size
        drop_keep_prob = config.drop_keep_prob
        pretrained_embed = config.pretrained_embed

        # 使用预训练好的模型进行词向量的编码的处理
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.embedding.weight.data.copy_(torch.from_numpy(pretrained_embed))
        self.embedding.weight.requires_grad = update_w2v
        # 定义卷积层
        self.conv = nn.Conv2d(1,num_filters,(kernel_size,embedding_dim))
        # 随即失活，防止过拟合
        self.dropout = nn.Dropout(drop_keep_prob)
        # 全连接
        self.FC2= nn.Linear(num_filters,512)
        self.fc = nn.Linear(512, n_class)


    def forward(self, x):
        x = x.to(torch.int64) #[32,50]
        x = self.embedding(x) #[32,50,50]
        x = x.unsqueeze(1)    #[32,1,50,50]
        x = F.relu(self.conv(x)).squeeze(3)   #[32,256,48]
        x = F.max_pool1d(x, x.size(2)).squeeze(2)  #[32,256]
        x = self.dropout(x)   #[32,256]
        x=self.FC2(x)         #[32,512]
        x=self.dropout(x)
        x = self.fc(x)        #[32,2]
        return x
```

模型的训练

```
def train(dataloader,train_dataset):

    # 初始化我们的模型
    model = TextCNN(config)
```

```
    if model_path:
        model.load_state_dict(torch.load(model_path))
    model.to(device)

    # 设置优化器和我们的损失函数
    optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)
    criterion = nn.CrossEntropyLoss()
    Loss_list=[]
    Train_acc=[]
    # 开始训练
    for epoch in range(epochs):
        train_acc = 0.0
        running_loss=0.0
        for batch_idx, (batch_x, batch_y) in enumerate(dataloader):
            batch_x, batch_y = batch_x.to(device), batch_y.to(device)
            output = model(batch_x)
            loss = criterion(output, batch_y)
            predict_y = torch.max(output, dim=1)[1]
            train_acc += torch.eq(predict_y,batch_y.to(device)).sum().item()
            running_loss += loss.item()
            if batch_idx % 200 == 0 & verbose:
                print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                    epoch+1, batch_idx * len(batch_x), len(dataloader.dataset),
                    100. * batch_idx / len(dataloader), loss.item()))

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        train_accurate =train_acc / len(train_dataset)
        print('[epoch %d] train_loss: %.3f  train_accuracy: %.3f' %
              (epoch + 1, running_loss / len(dataloader),train_accurate))


        Loss_list.append(running_loss / len(dataloader))
        Train_acc.append(train_accurate)

    #绘制训练过程中的损失和准确率曲线
    torch.save(model.state_dict(), 'model.pth')
    print('Finished Training')
    print("开始绘制训练曲线：----------------------------")
    x1 = np.arange(1, len(Loss_list) + 1)
    plt.plot(x1, Loss_list, '--', color='red', label='Train_loss')
    plt.title('model losses')
    plt.ylabel('Loss')
    plt.xlabel('epoch')
    plt.legend()
    plt.show()

    x1 = np.arange(1, len(Train_acc) + 1)
    plt.plot(x1, Train_acc, 'o-', color='blue', label='Train_acc')
    plt.title('model acc')
    plt.ylabel('acc')
```

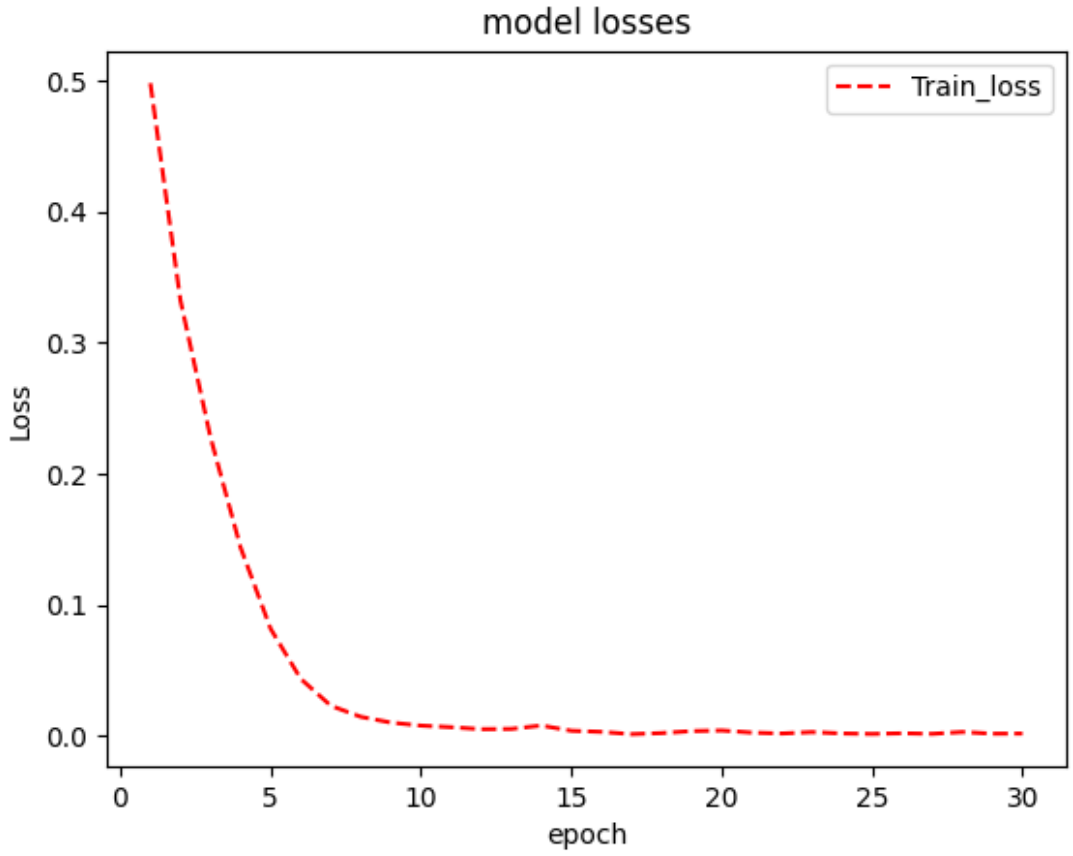
```
    plt.xlabel('epoch')
    plt.legend()
    plt.show()
```

预测函数：验证在测试集上的分类的效果：

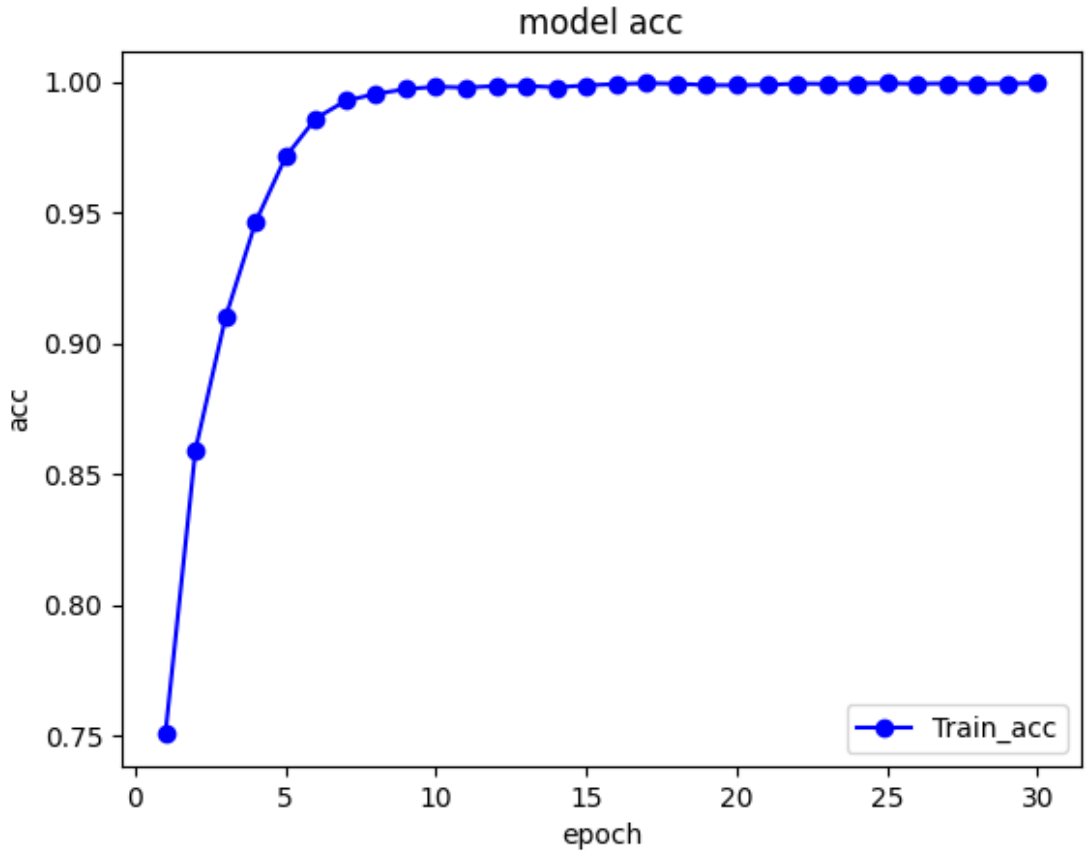
```
def predict(dataloader):

    #加载训练好的模型
    model = TextCNN(config)
    model.load_state_dict(torch.load(model_path))
    model.eval()
    model.to(device)

    # 测试
    count, correct, real_predict_00, real_predict_01, real_predict_10,
real_predict_11 = 0, 0, 0, 0, 0, 0
    for _, (batch_x, batch_y) in enumerate(dataloader):
        batch_x, batch_y = batch_x.to(device), batch_y.to(device)
        output = model(batch_x)
        count += len(batch_x)
        correct += (output.argmax(1) == batch_y).float().sum().item()
        real_predict_00 += np.array([(output.argmax(1)[idx] == 0 and batch_y[idx] ==
0).float().cpu().numpy() for (idx, _) in enumerate(batch_y)]).sum().item()
        real_predict_01 += np.array([(output.argmax(1)[idx] == 0 and batch_y[idx] ==
1).float().cpu().numpy() for (idx, _) in enumerate(batch_y)]).sum().item()
        real_predict_10 += np.array([(output.argmax(1)[idx] == 1 and batch_y[idx] ==
0).float().cpu().numpy() for (idx, _) in enumerate(batch_y)]).sum().item()
        real_predict_11 += np.array([(output.argmax(1)[idx] == 1 and batch_y[idx] ==
1).float().cpu().numpy() for (idx, _) in enumerate(batch_y)]).sum().item()

    # calculate accuracy, precision, recall, F1_score, confusion_matrix
    accuracy = correct/count
    precision = real_predict_00 / (real_predict_00 + real_predict_10)
    recall = real_predict_00 / (real_predict_00 + real_predict_01)
    F1_score = 2*precision*recall/(precision+recall)
    confusion_matrix = [[real_predict_00, real_predict_01], [real_predict_10,
real_predict_11]]
    print('The accuracy, precision, recall, F1_score, confusion_matrix of test
is\n{:.2f}% \n{} \n{} \n{} \n{}.'.format(100*accuracy, precision, recall, F1_score,
confusion_matrix))
```

实验结果：

迭代三十次的模型的损失图：



迭代三十次模型的准确率变化曲线：



模型迭代到第三十次的时候，损失0.011，准确率达到了0.997，达到了目标的预测

```
[epoch 30]:  train_loss: 0.011  train_accuracy: 0.997
```

使用predict预测评估：

测试集上的效果：

acc	precision	recall	F1
84.11%	0.8626373626373627	0.7929292929292929	0.82631578947368434

通过以上实验，达到实验要求.